SDCTune: A Model for Predicting the SDC Proneness of an Application for Configurable Protection

Qining Lu, Karthik Pattabiraman
University of British Columbia (UBC)

Jude Rivers, Meeta Gupta
IBM Research T.J. Watson
Motivation: Transient Errors

Particle strikes, temperature, etc.,

Transient hardware faults

Source: Feng et. al., ASPLOS’2010

Transient hardware errors (aka. Soft errors) *increase* as feature sizes *shrink*
Motivation: Application-level Techniques

Only a fraction of the errors at the circuit level impacts the application.

More economical to deploy techniques at application.

Impactful Errors
Silent Data Corruption (SDC): Our focus in this paper

Example: Bfs

Wrong output

Correct output

Results lost:
Our Goals

• **Detect** Silent Data Corruption (SDC)

• **High Coverage with Low** Overhead

• **Configurable** protection overhead

Selectively protect highly SDC-prone variables in program
Traditional approaches Vs. Our approach

Traditional

Fault injection

Few lead to SDCs

Thousands of runs of the application

... 

Time consuming (runs application thousands of times)

Need to manually choose variables to protect

Ours

Program code

Static and dynamic program analysis

Selected variables

Protect/duplicate Selected variables

Performance overhead budget

Time saving (dynamic analysis only runs the application once)

Automatically choose variables to protect subject to performance
Fault model

- Single bit flip fault
- One fault per run
- Errors in registers and execution units
- Program data that is visible at architectural level
• Motivation and Goal
• Approach
• Evaluation and Results
• Conclusion
Overall Approach

- Step 1: Perform fault injections to understand SDC characteristics of code constructs
- Step 2: Heuristics identifying code regions prone to SDC causing faults
- Step 3: SDCTune model building and protection
Initial study: Goals

- **Initial fault injection experiments**
  - The goal is to understand the reasons for SDC failures
  - Used to formulate heuristics for selective protection

- **Manually inspect why SDC occurs**
  - Highly executed instructions cover most SDCs
  - Not all highly executed instructions should be protected
  - Find common patterns used for developing heuristics
Initial Study: Method

- Performed using LLFI, high level fault injector validated for SDC-causing errors [DSN’14]
Initial study: Findings

• SDC proneness of instruction depends on:
  • The fault propagation in its data dependency chain
  • The SDC proneness of the end point of that chain

• End points of data dependency chain:
  • Store operations
  • Comparison operations

Need heuristics for fault propagation, store operations, comparison operations
Heuristics: Fault propagation

HP1: The SDC proneness of an instruction will decrease if its result is used in either fault masking or crash prone instructions.

Fault occurs

Corrupted bits

Corrupted variable

Trunc operation

Result variable

Fault masked

Correct output
Heuristics: Store operations

<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
<th>Major related features</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addr NoCmp</td>
<td>The stored value is used in calculating memory addresses but not comparison results</td>
<td>Data width</td>
</tr>
<tr>
<td>Addr Cmp</td>
<td>The stored value is used in calculating both memory addresses and comparison results</td>
<td>Data width and control flow deviation</td>
</tr>
<tr>
<td>Cmp NoAddr</td>
<td>The stored value is used in calculating comparison results but not memory addresses</td>
<td>Resilient or Unresilient comparison</td>
</tr>
<tr>
<td>NoCmp NoAddr</td>
<td>The stored value is neither used in memory address calculation nor comparison results</td>
<td>Used in output or not</td>
</tr>
</tbody>
</table>

HS1: Addr NoCmp stored values have low SDC proneness in general
HS2: Addr Cmp stored values have higher SDC proneness than Addr NoCmp

<More heuristics in paper>
Heuristics: Comparison operations

HC1: Nested loop depths affect the SDC proneness of loops’ comparison operations.

```c
void BZ2_hbMakeCodeLengths
(...){
    while(nHeap>1){ //outer loop
        ...
        while(weight[tmp]<weight[
            heap[zz>>1]]){
            //inner loop
            Heap[zz]=heap[zz>>1];
            zz>>=1;
        }
    }
}
```

SDC proneness of “`nHeap>1`” higher than “`weight[tmp]<weight[heap[zz>>1]]`”

<More heuristics in paper>
SDCTune: Build model

- **Classification**
  - *Different types of usage are usually independent of each other*
  - *Classify the stored values and comparison values according to the heuristic features we observed before*

- **Regression**
  - *With same type of usages, SDC rate may show gradually correlations to several features*

52 features in total used in the model
SDCTune: Example model

Example: *tree structure for Store*
SDCTune: Selection Algorithm

1. Application Source Code
2. Compiler
3. IR
4. SDCTune
5. Selection Algorithm
6. Performance Overhead
7. Representative inputs
8. Data Variables or Locations to Protect
9. Backward slice replication

Initial Study → Heuristics → SDCTune
SDCTune: Optimizations

Adding the instructions to the protection set to save checkers

Move checker out of loop body
• Motivation and Goal
• Approach
• Evaluation and Results
• Conclusion
Evaluation: Work Flow

**Training phase**
- SDC rate for each instruction $P(SDC|I)$ from training programs
- Features extracted based on heuristic knowledge from training programs
- Training (Regression) $P(SDC|I)$ Predictor

**Testing and using phase**
- Optimal selection: est. $P(SDC|I)P(I)$ vs. $P(I)$
- Set $\{\text{Instructions}\}$ for a certain overhead bound ($\sum P(I)$)
- Random Fault Injection Results from testing programs

**Measure real coverage on testing programs**
- Actual SDC coverage for testing programs
- Features extracted from testing programs
Evaluation: Work Flow

- SDC rate for each instruction $P(\text{SDC}|I)$ from \textit{training programs}
- Training (Regression)
- Features extracted based on heuristic knowledge from \textit{training programs}
- Optimal selection: est. $P(\text{SDC}|I)P(I)$ vs. $P(I)$
- Set\{Instructions\} for a certain overhead bound ($\sum P(I)$)
- Random Fault Injection Results from \textit{testing programs}
- Features extracted from \textit{testing programs}
- Actual SDC coverage for \textit{testing programs}
### Evaluation: Benchmarks

<table>
<thead>
<tr>
<th>Training programs</th>
<th>Testing programs</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Program</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td>IS</td>
<td>Integer sorting</td>
</tr>
<tr>
<td>LU</td>
<td>Linear algebra</td>
</tr>
<tr>
<td>Bzip2</td>
<td>Compression</td>
</tr>
<tr>
<td>Swaptions</td>
<td>Price portfolio of swaptions</td>
</tr>
<tr>
<td>Water</td>
<td>Molecular dynamics</td>
</tr>
<tr>
<td>CG</td>
<td>Conjugate gradient</td>
</tr>
</tbody>
</table>
Evaluation: Experiments

- Estimate overall SDC rates using SDCTune and compare with fault injection experiments
  - Measure correlation between predicted and actual

- Measure SDC Coverage of detectors inserted using SDCTune for different overhead bounds
  - Consider 10, 20 and 30% performance overheads

- Compared performance overhead and efficiency with full duplication and hot-path duplication
  - Efficiency = SDC coverage / Performance overhead
**Results: Overall SDC Rates**

<table>
<thead>
<tr>
<th></th>
<th>Training programs</th>
<th>Testing programs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rank correlation*</td>
<td>0.9714</td>
<td>0.8286</td>
</tr>
<tr>
<td>P-value**</td>
<td>0.00694</td>
<td>0.0125</td>
</tr>
</tbody>
</table>

The graph shows a comparison between the rank of overall SDC rates by estimation and the rank of overall SDC rates by fault injection experiment for both training and testing programs. The correlation coefficient and P-value indicate a strong positive correlation and statistically significant results respectively.
Results: SDC Coverage

<table>
<thead>
<tr>
<th>Overhead</th>
<th>Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>10%</td>
<td>44.8%</td>
</tr>
<tr>
<td>20%</td>
<td>78.6%</td>
</tr>
<tr>
<td>30%</td>
<td>86.8%</td>
</tr>
</tbody>
</table>

Training programs:

Testing programs:

<table>
<thead>
<tr>
<th>Overhead</th>
<th>Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>10%</td>
<td>39%</td>
</tr>
<tr>
<td>20%</td>
<td>63.7%</td>
</tr>
<tr>
<td>30%</td>
<td>74.9%</td>
</tr>
</tbody>
</table>
Results: Full Duplication Overheads

Full duplication and hot-path duplication (top 10% of paths) have high overheads. For full duplication it ranges from 53.7% to 73.6%, for hot-path duplication it ranges from 43.5 to 57.6%.
Results: Detection Efficiency

<table>
<thead>
<tr>
<th>Normalized Detection Efficiency</th>
<th>10% overhead</th>
<th>20% overhead</th>
<th>30% overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Training programs</td>
<td>2.38</td>
<td>2.09</td>
<td>1.54</td>
</tr>
<tr>
<td>Testing programs</td>
<td>2.87</td>
<td>2.34</td>
<td>1.84</td>
</tr>
</tbody>
</table>
• Motivation and Goal
• Approach
• Evaluation and Results
• Conclusion
Conclusion and Future Work

• Configurable protection techniques for SDC failures are required as transient fault rates increase

• We find heuristics to estimate SDC proneness for program variables based on static and dynamic features

• SDCTune model to guide configurable SDC protection
  • Accurate at predicting relative SDC rates of applications
  • Much better detection efficiency compared to full duplication

• Future work
  • Improving the model’s accuracy using auto-tuning
  • Using symptom based detectors for protection

http://blogs.ubc.ca/karthik/